frama C
Software Analyzers

1st Asian-Pacific Summer School on Formal Methods
Course 12: Deductive verification of C programs with Frama-C and Jessie

Virgile Prevosto

CEA List

August 30, 2009

frama C
Software Analyzers

outline

frama C
Software Analyzers

Jessie Usage

Function Contracts

Advanced Specification

- Example 1: Searching
- Example 2: Sorting

frama C
Software Analyzers

What is Jessie?

- Hoare-logic based plugin, developed at INRIA Saclay.
- Input: a program and a specification
- Jessie generates verification conditions
- Use of Automated Theorem Provers to discharge the VCs
- If all VCs are proved, the program is correct with respect to the specification
- Otherwise: need to investigate why the proof fails
  - Fix bug in the code
  - Adds additional annotations to help ATP
  - Interactive Proof (Coq)

frama C
Software Analyzers

What is Jessie Useful for?

Usage

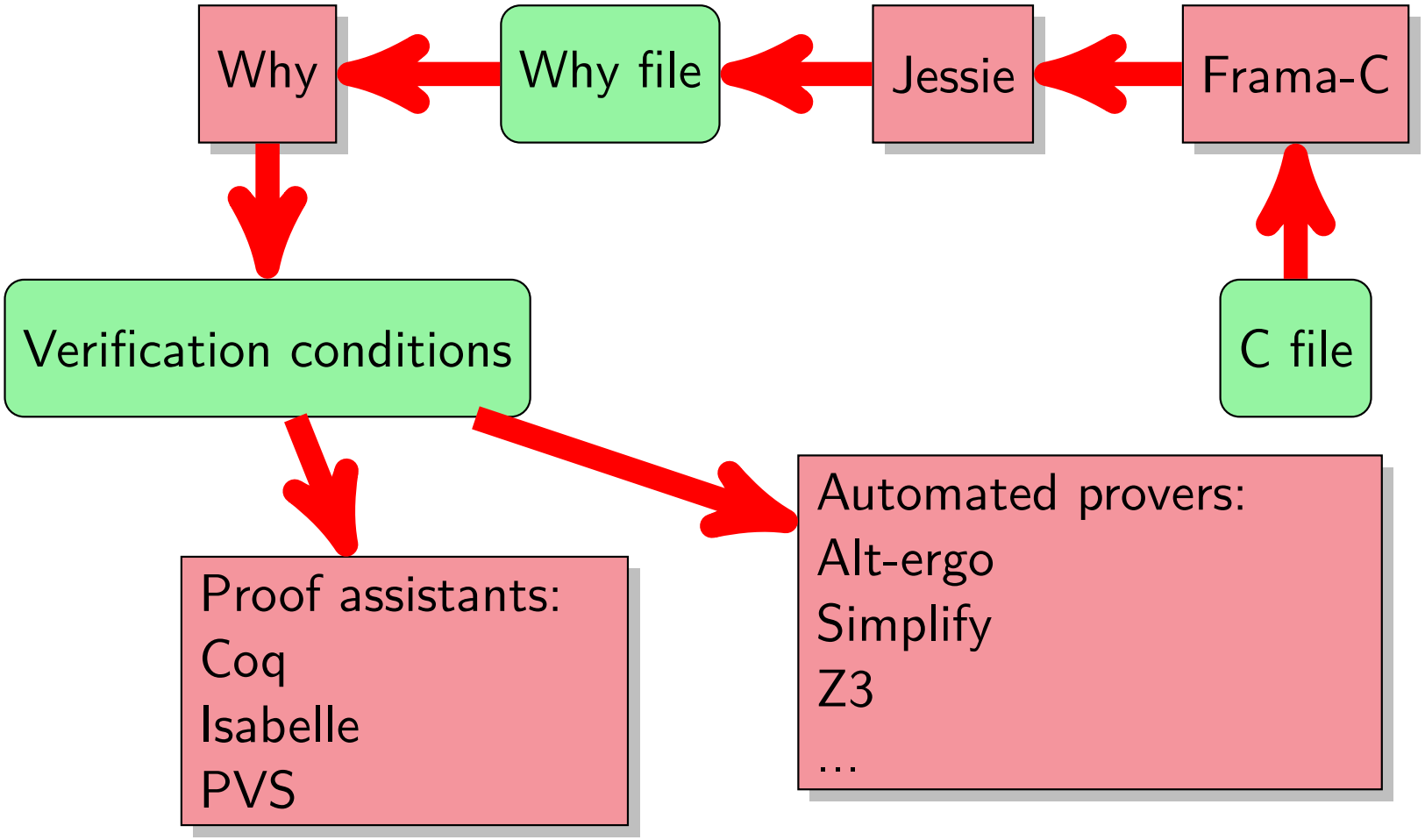
- Proof of functional properties of the program
- Modular verification (function per function)

Limitations

- Cast between pointers and integers
- Limited support for union type
- Aliasing requires some care

frama C
Software Analyzers

From Frama-C to Theorem Provers



frama C
Software Analyzers

A first example

Check safety of a function

- Pointer accesses
- Arithmetic overflow
- Division

```
unsigned int M;

void mean(unsigned int* p, unsigned int* q) {
  M = (*p + *q) / 2;
}
```

frama C
Software Analyzers

A first specification

```
unsigned int M;

/*@ requires \valid(p) ∧ \valid(q);
 */
void mean(unsigned int* p, unsigned int* q) {
  if (*p ≥ *q) { M = (*p − *q) / 2 + *q; }
  else { M = (*q − *p) / 2 + *p; }
}
```

frama C
Software Analyzers

Jessie Usage

Function Contracts

Advanced Specification
- Example 1: Searching
- Example 2: Sorting

frama C
Software Analyzers

ACSL

Safety of a program is important, but this is not sufficient: We also want it to do “the right thing”...
But in order for jessie to verify that, we need to explain it what “the right thing” is, and to explain it formally
This is the purpose of ACSL, ANSI/ISO C Specification Language.

- Behavioral specification language à la JML and Eiffel
- Function contracts
- Logic models
- Independent from any plug-in

frama C
Software Analyzers

Behavior of a function

- Functional specification
- Pre-conditions (`requires`)
- Post-conditions (`ensures`)

Example

```
unsigned int M;
/*@
  requires \valid(p) ∧ \valid(q);
  ensures M ≡ (*p + *q) / 2;
*/
void mean(unsigned int* p, unsigned int* q) {
  if (*p ≥ *q) { M = (*p − *q) / 2 + *q; }
  else { M = (*q − *p) / 2 + *p; }
}
```

frama C
Software Analyzers

Logic function

One can define logic functions as follows:

```
/*@ logic integer mean(integer x, integer y) =
          (x+y)/2; */
```

frama C
Software Analyzers

Side effects

The specification:

```
/*@
  requires \valid(p) ∧ \valid(q);
  ensures M ≡ (*p + *q) / 2;

*/
void mean(unsigned int* p, unsigned int* q);
```

An admissible implementation:

```
void mean(int *p, int* q) {
  *p = *q = M = 0; }
```

Side effects

The specification:

```
/*@
   requires \valid(p) ∧ \valid(q);
   ensures M ≡ (*p + *q) / 2;
   ensures *p ≡ \old(*p) ∧ *q ≡ \old(*q);
*/
void mean(unsigned int* p, unsigned int* q);
```

An admissible implementation:

```
void mean(int *p, int* q) {
   if (*p ≥ *q) ... else ...
   A = 0; }
```

frama C
Software Analyzers

Side effects

The specification:

```
/*@
   requires \valid(p) ∧ \valid(q);
   ensures M ≡ (*p + *q) / 2;
   assigns M;
*/
void mean(unsigned int* p, unsigned int* q);
```

An admissible implementation:

```
void mean(int *p, int* q) {
   if (*p ≥ *q) { M = (*p − *q) / 2 + *q; }
   else { M = (*q − *p) / 2 + *p; }
}
```

frama C Software Analyzers

Termination

- Post condition true *when the function exits normally.*
- By default, a function always terminates...
- ... as long as its pre-condition holds.

```
/*@
  requires \valid(p) ∧ \valid(q);
  ensures ...
  assigns M;
  terminates \true;
*/
void mean(int* p, int* q) {

  if (p ≡ NULL ∨ q ≡ NULL) while(1);
  if (*p ≥ *q) ...
}
```

frama C
Software Analyzers

Jessie Usage

Function Contracts

Advanced Specification

- Example 1: Searching
- Example 2: Sorting

frama C
Software Analyzers

A concrete example

Informal spec

- Input: a sorted array and its length, an element to search.
- Output: index of the element or -1 if not found

Implementation

```
int find_array(int* arr, int length, int query) {
  int low = 0;
  int high = length - 1;
  while (low ≤ high) {
    int mean = low + (high -low) / 2;
    if (arr[mean] ≡ query) return mean;
    if (arr[mean] < query) low = mean + 1;
    else high = mean - 1;
  }
  return -1;
}
```

frama C Software Analyzers

Predicate definition

What does "sorted" mean?

```
/*@
  predicate sorted{L}(int* arr, integer length) =
  ∀ integer i,j; 0≤i≤j<length ⟹ arr[i] ≤ arr[j];
*/
```

ACSL predicates

- Give a formal definition of the properties of objects
- Can be used in annotations
- Must be tied to a program point when performing a memory access
- Some predicate can relate two or more program points (see after)

Function behaviors

`find_array` has two distinct behaviors, depending on whether the query is in the array or not. This can be reflected in the contract in the following way:

```
/*@
  behavior exists:
  assumes ∃ integer i;
          0≤i<length ∧ arr[i] ≡ query;
  ensures arr[\result] ≡ query;

  behavior not_exists:
  assumes ∀ integer i;
          0≤i<length ⟹ arr[i] ≢ query;
  ensures \result ≡ −1;
*/
```

frama C Software Analyzers

Loop invariant

Role of the loop invariant

- Must be inductive (if it holds at the beginning, then it holds at the end)
- Capture the effects of one loop step
- Represents the only things known at the exit of the loop
- Must be strong enough to allow to derive the post-condition

Example

```
/*@
  loop invariant 0 ≤ low;
  loop invariant high < length;
  loop invariant ∀ integer i;
    0 ≤ i < low ⟹ arr[i] < query;
  loop invariant ∀ integer i;
    high < i < length ⟹ arr[i] > query;
*/
```

frama C
Software Analyzers

Loop variant

Total correctness

- Needed for proving termination
- Expression which strictly decreases at each step
- And stay non-negative

Example

```
/*@ loop invariant ...
    loop variant high - low;
*/
```

frama C Software Analyzers

Assertions

Usage

- A property which must hold at a given point
- Allows to guide the automated provers
- Can be associated to a particular behavior

Example

```
int mean = low + (high −low) / 2;
//@ assert low ≤ mean ≤ high;
 if (arr[mean] ≡ query)
 //@ for not_exists: assert \false;
```

frama C
Software Analyzers

An example

Informal specification

- Input: an array and its length
- Output: the array is sorted in ascending order

```
int index_min(int* a, int low, int high);

void swap(int* arr, int i, int j);

void min_sort(int* arr, int length) {
  for(int i = 0; i < length; i++) {
    int min = index_min(arr,i,length);
    swap(arr,i,min);
  }
}
```

frama C
Software Analyzers

Function Calls

- Post-conditions and assigns are the only things that the caller knows when the callee returns
- Caller must fulfill the pre-condition of callee before the call

frama C Software Analyzers

Inductive Predicates

- Case definition: $H_i \Rightarrow Pred$
- Horn clause: *Pred* can only appear positively in H_i (ensures consistency)
- Smallest fixpoint: predicate holds iff one of the H_i holds

```
/*@ inductive Permut{L1,L2}
      (int a[], integer l, integer h) {
    case Permut_trans{L1,L2,L3}:
      ∀ int a[], integer l, h;
        Permut{L1,L2}(a, l, h) ∧
        Permut{L2,L3}(a, l, h) ⟹
          Permut{L1,L3}(a,l, h) ;
    case Permut_swap{L1,L2}: ...
   }
@*/
```

Axiomatics

It is also possible to have axiomatic definitions of predicates and logic functions:

```
/*@ axiomatic Permut{
predicate permut{L1,L2}(int* arr, integer length);
axiom perm_refl{L}:
 ∀ int* arr, integer length; permut{L,L}(arr,length);
axiom perm_swap{L1,L2}:
 ∀ int* arr, integer length,i,j;
  0≤i<length ∧ 0≤ j < length ⟹
  swap{L1,L2}(arr,i,j) ⟹ permut{L1,L2}(arr,length);
axiom perm_trans{L1,L2,L3}: ∀ int* arr;
 ∀ integer length; permut{L1,L2}(arr,length) ⟹
  permut{L2,L3}(arr,length) ⟹
  permut{L1,L3}(arr,length);}*/
```